UNIVERSITY OF ALBERTA

Dataflow Analysis

CMPUT 620 — Static Program Analysis
Karim Ali

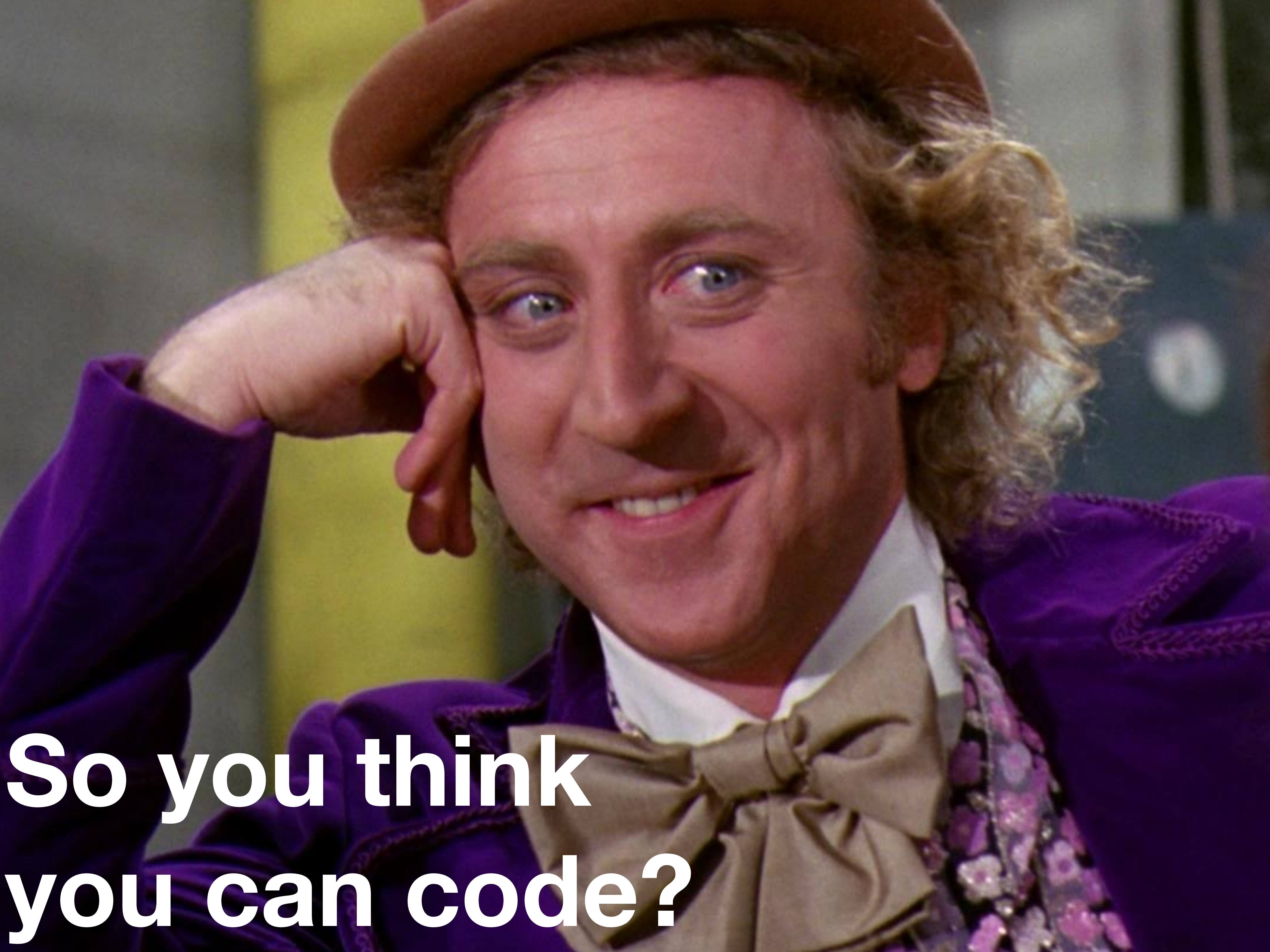
September 12, 2017
GSB 8-59

Disclaimer

- Some of the material presented here is based on the slides from CS 744 by Ondřej Lhoták at the University of Waterloo, and DECA course by Eric Bodden at TU Darmstadt

Today's Lecture

- Sample Static Analyses
- Intermediate Representations
- Lattices
- Dataflow Analysis Framework


So you think
you can code?

What's the output?

```
System.out.println("Hello, World!");
```

How about now?

```
if(arbitraryComputation()) {
  System.out.println(“Hello, World!”);
} else {
  System.out.println(“Goodbye”);
}
```

Any errors?

```
if(arbitraryComputation()) {
  int a[] = new int[6];
  a[10] = 10;
}
```

"For **any** interesting property *Pr* of the behaviour of a program, it is **impossible** to write an analysis that can decide for every program *p* whether *Pr* holds for *p*."

–Rice's Theorem

Static analysis by definition is **undecidable**

So we're doomed!

Not really…

Static Analysis

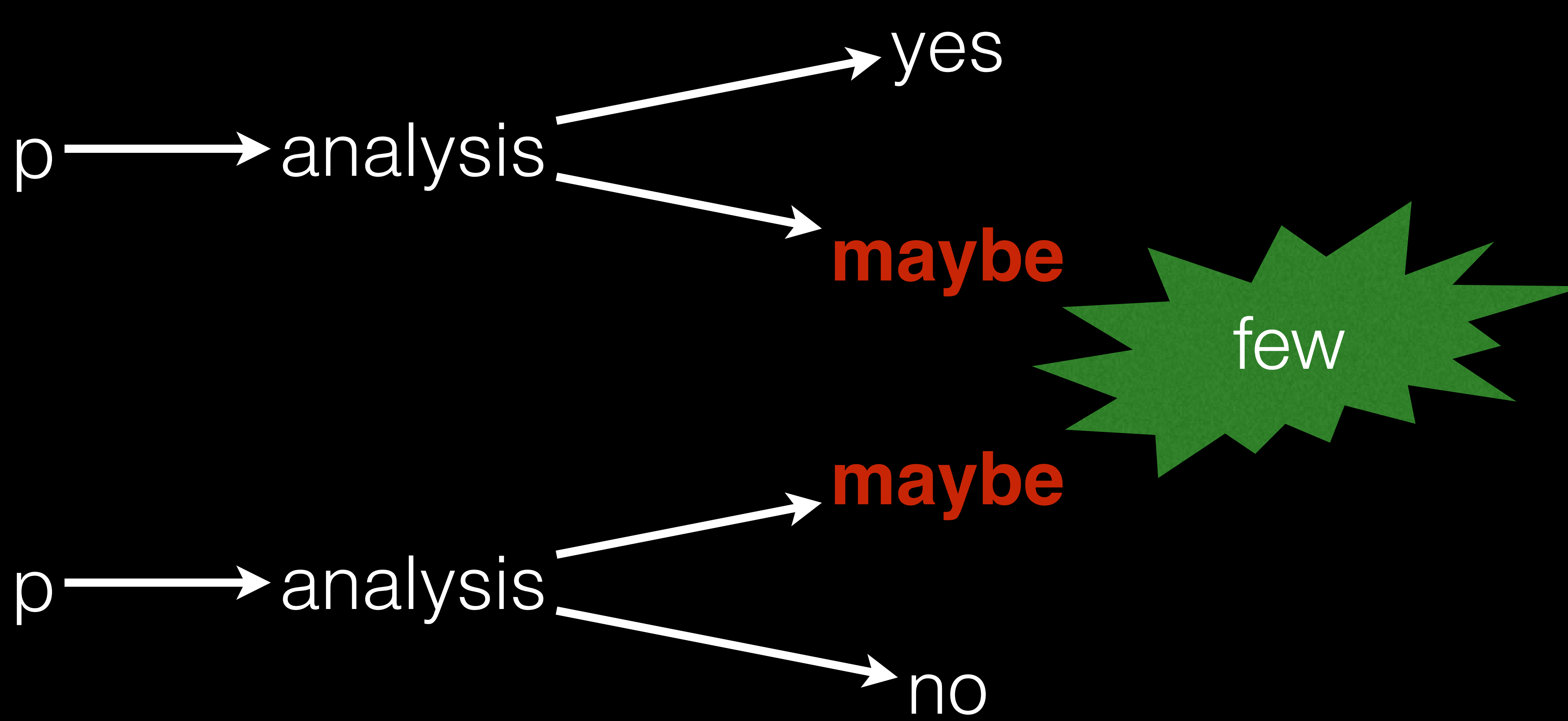
- Settle for an approximation of *Pr*
- Make it as “good” as possible

$$p \longrightarrow \text{analysis} \begin{cases} \nearrow \text{yes} \\ \searrow \text{maybe} \end{cases}$$

$$p \longrightarrow \text{analysis} \begin{cases} \nearrow \text{maybe} \\ \searrow \text{no} \end{cases}$$

Static Analysis

- Settle for an approximation of *Pr*
- Make it as “good” as possible



Sample Analyses

Constant Propagation

```
a = 1;
b = 2;
c = a + b;
```

```
a = 1;
b = 2;
c = 3;
```

Reduces # calculations

Dead Code Elimination

```
if(DEBUG) {
  println("...");
}
```

Reduces app size

Typestate Analysis

```
File a = new File();
a.open();

File b = new File();

if(...) b = a;

b.close();
```

Detect resource leaks

Intermediate Representations

Intermediate Representations

Source Code

↓

High-level IR

↓

Medium-level IR

↓

Low-level IR

↓

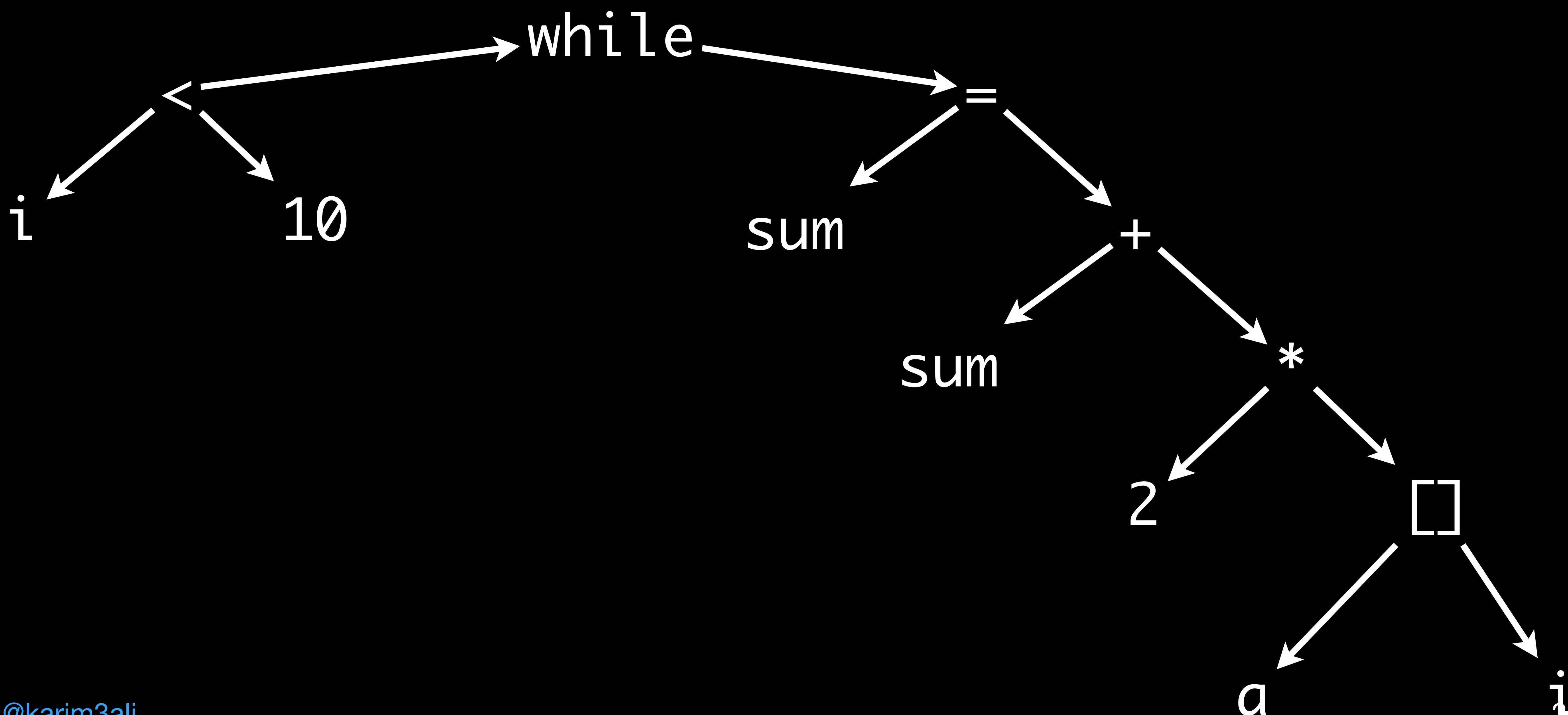
Target Code

Intermediate Representations

```
while(i < 10) { sum = sum + 2 * a[i]; }
```

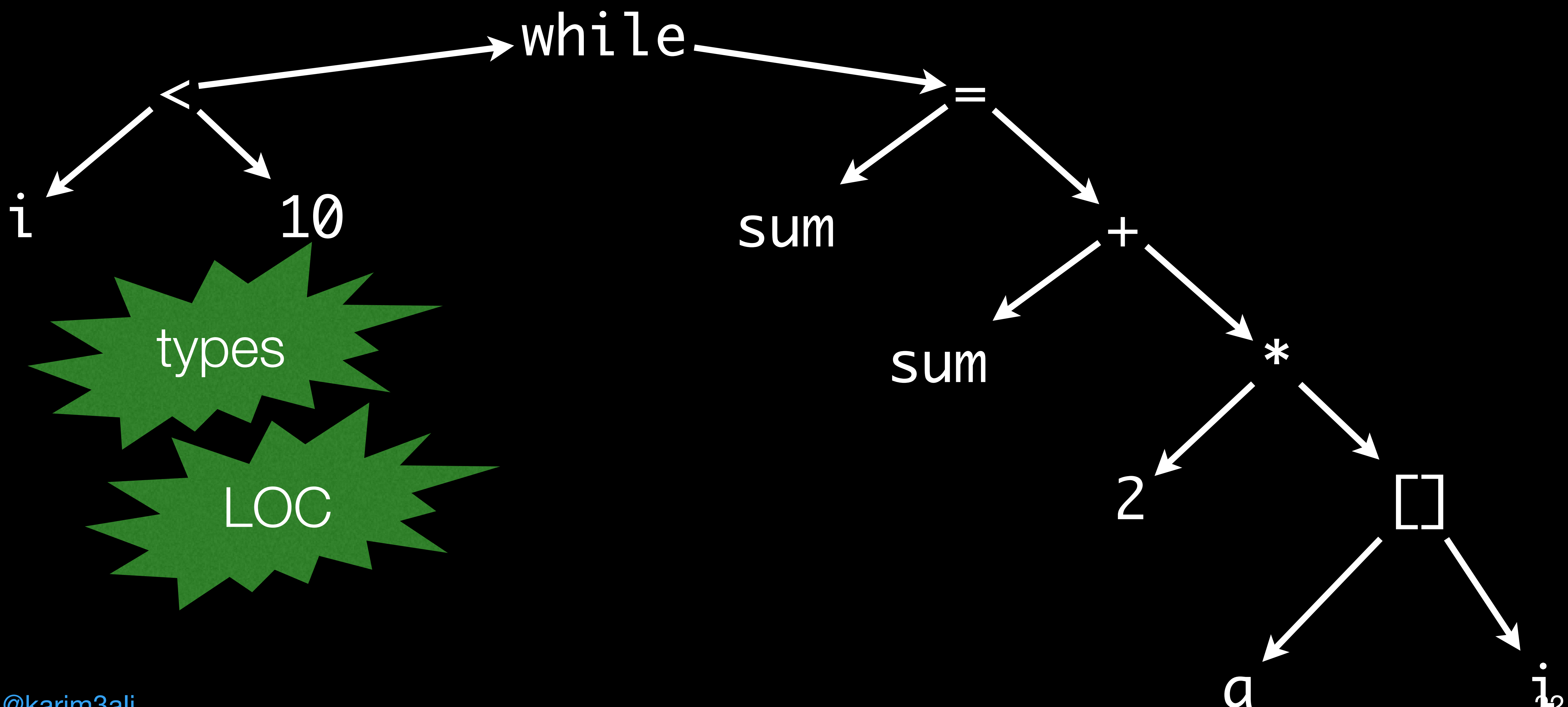
Abstract Syntax Tree

```
while(i < 10) { sum = sum + 2 * a[i]; }
```



Annotated AST

```
while(i < 10) { sum = sum + 2 * a[i]; }
```



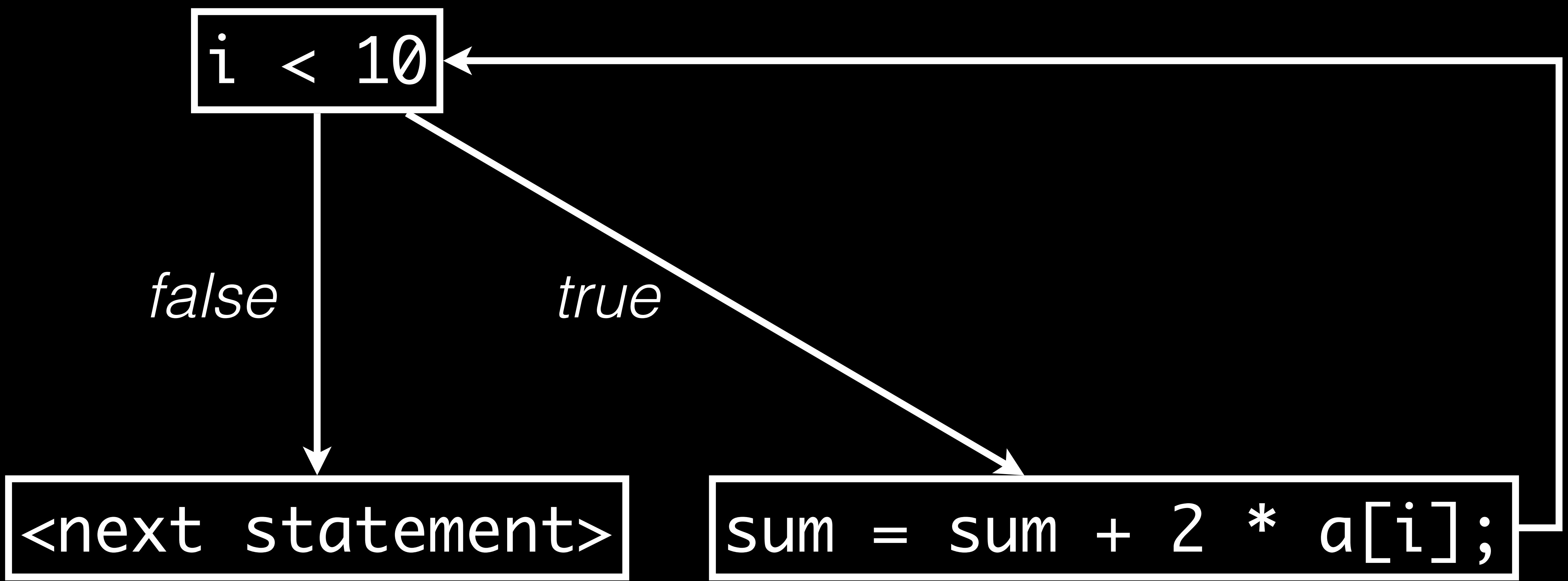
3-Address Code

```
while(i < 10) { sum = sum + 2 * a[i]; }
```

```
L0:
  t1 = i >= 10;
  if t1 goto L1;
  t2 = i * 4;
  t3 = a + t2;
  t4 = *t3;
  t5 = 2 * t4;
  sum = sum + t5;
  goto L0;
L1:
```

Control-Flow Graph

```
while(i < 10) { sum = sum + 2 * a[i]; }
```



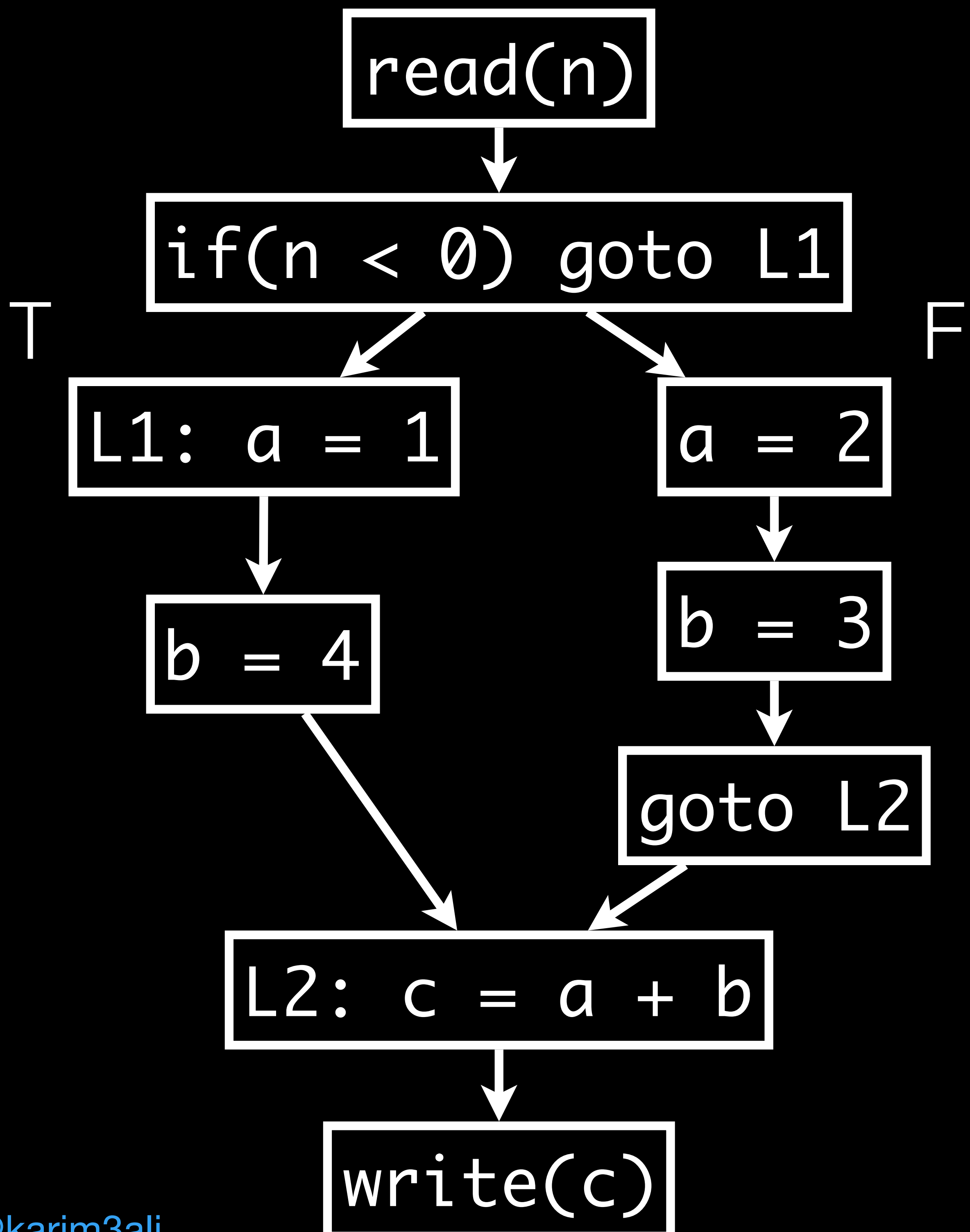
IR Tradeoffs

High-Level IR	Low-Level IR
language-specific	language-independent
machine-independent	machine-specific
tree/graph	instruction sequence
control-flow	gotos
compound expressions	simple expressions
high-level constructs	expanded constructs

Let’s consider this code

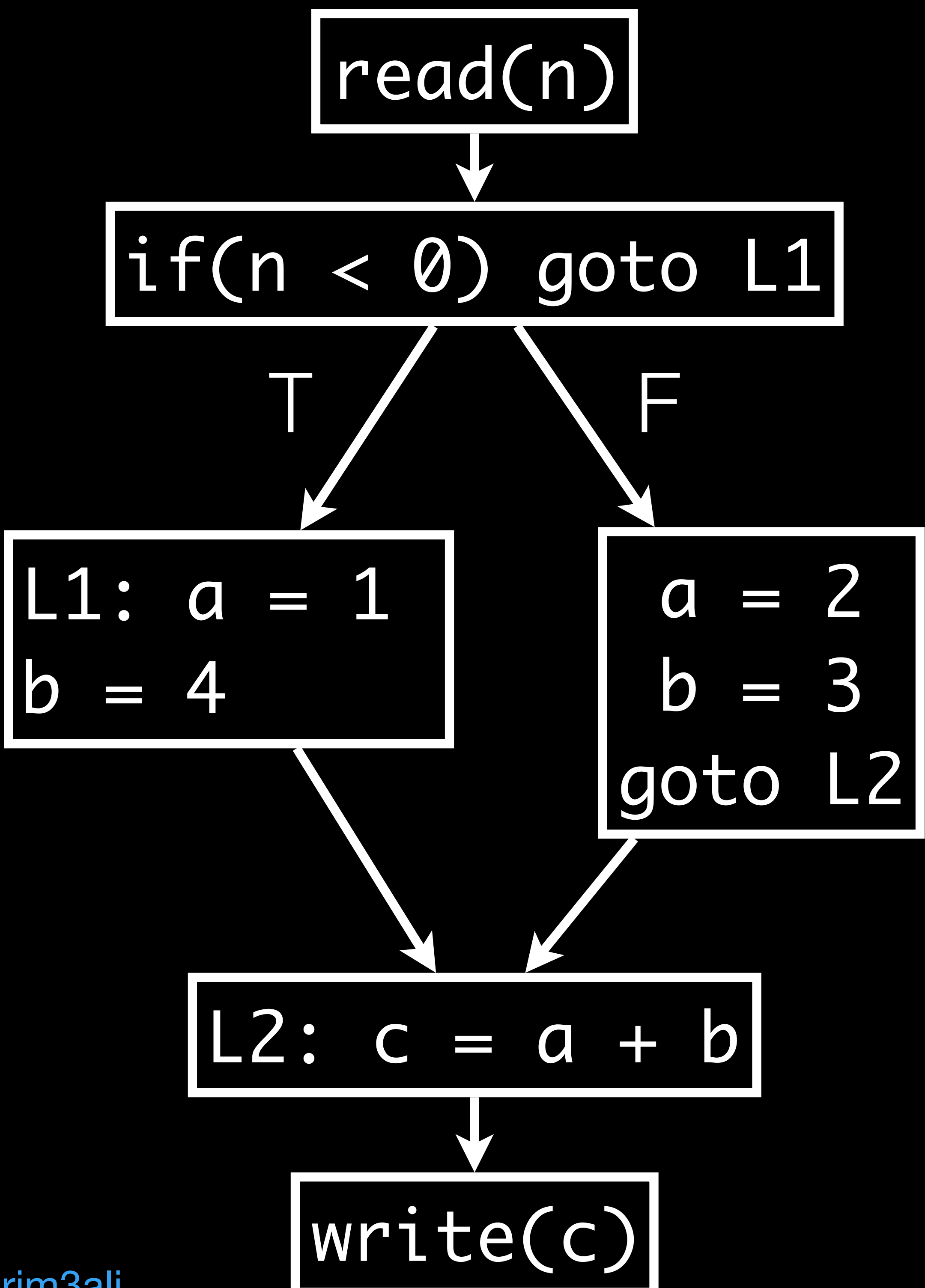
```
read(n);

if(n < 0) {
  a = 1;
  b = 4;
} else {
  a = 2;
  b = 3;
}

c = a + b;
write(c);
```

Control-Flow Graph



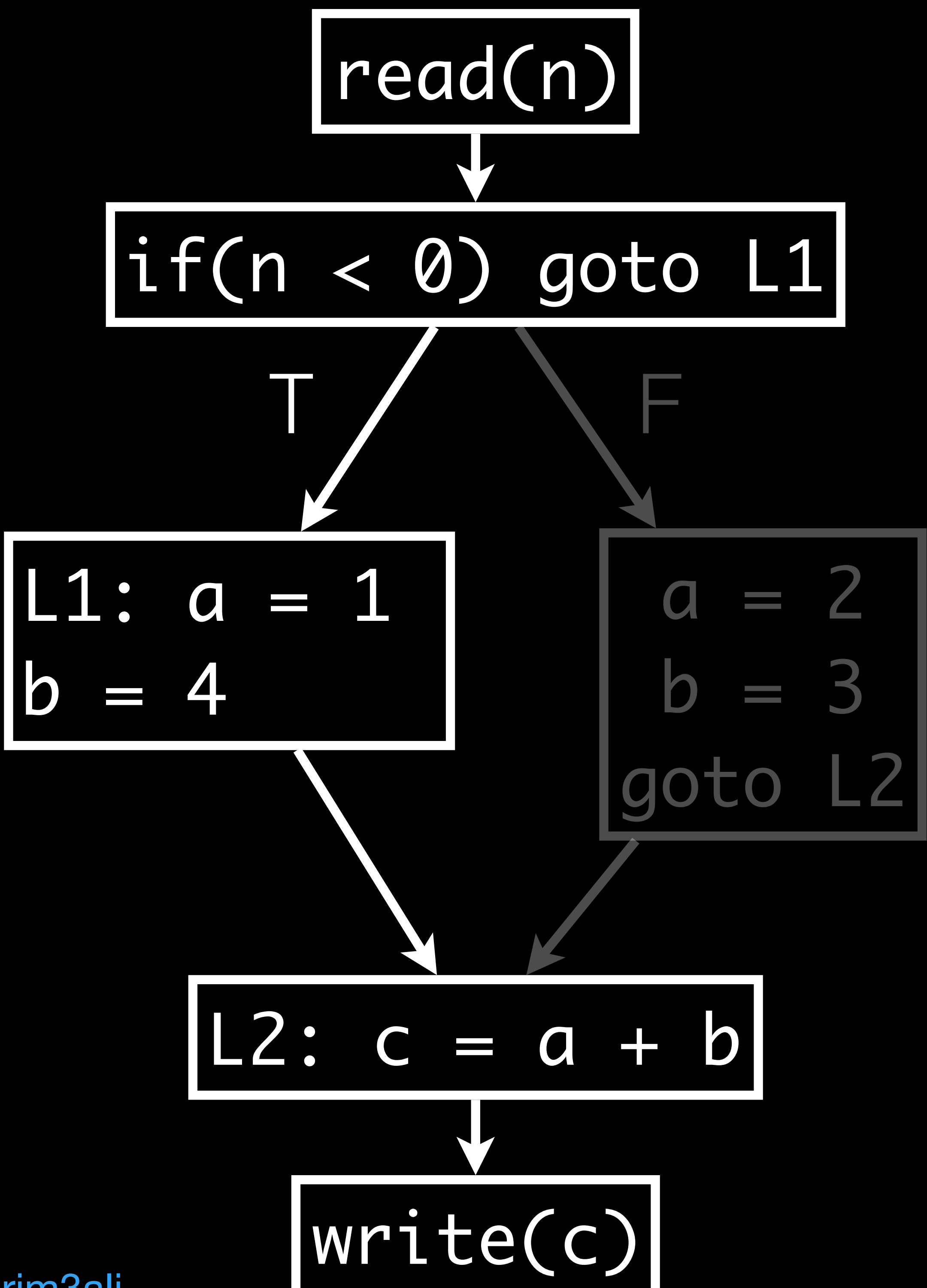

```
read(n);

if(n < 0) {
  a = 1;
  b = 4;
} else {
  a = 2;
  b = 3;
}

c = a + b;
write(c);
```

Basic-Block Graph



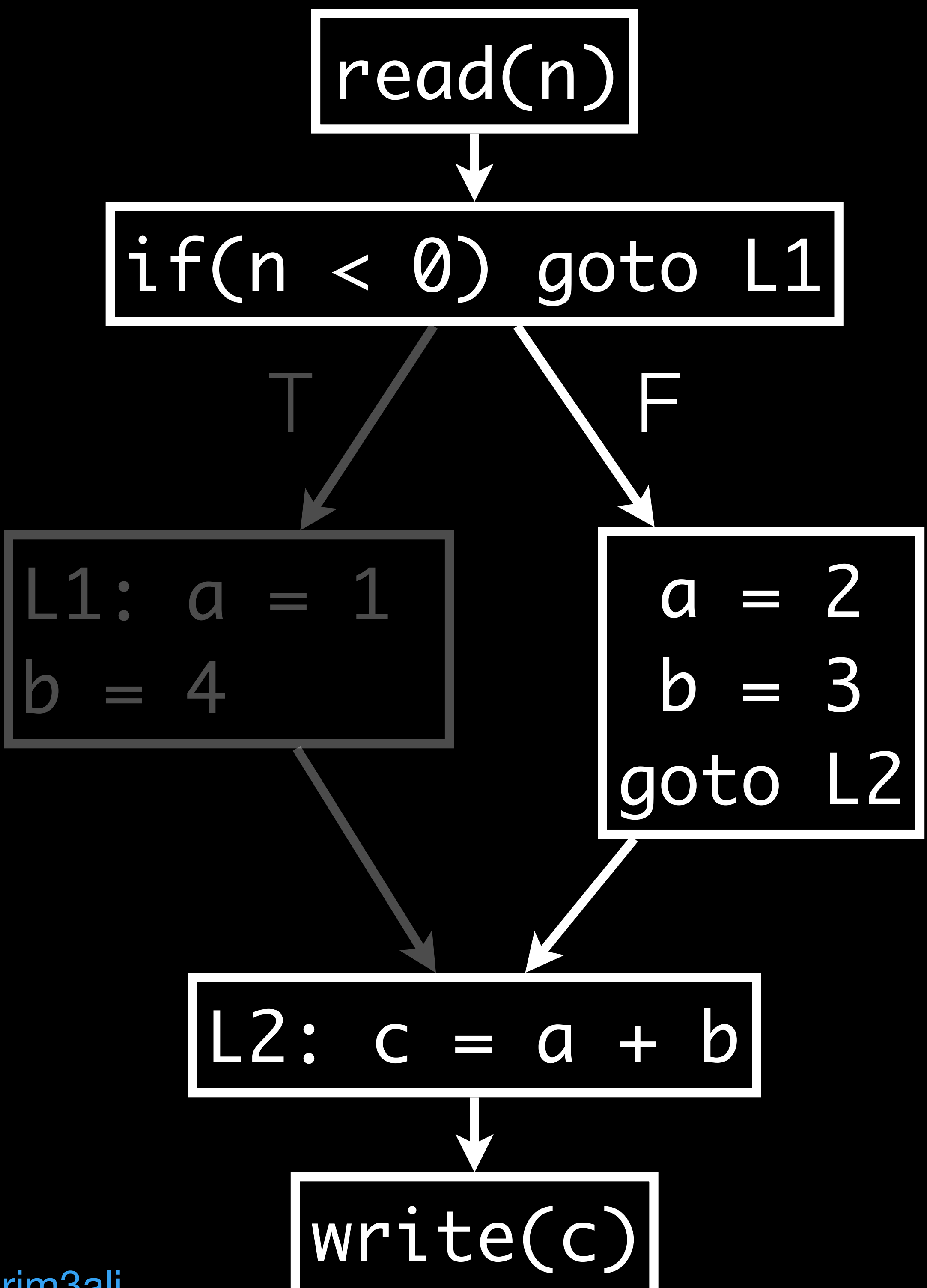

```
read(n);

if(n < 0) {
  a = 1;
  b = 4;
} else {
  a = 2;
  b = 3;
}

c = a + b;
write(c);
```

A path



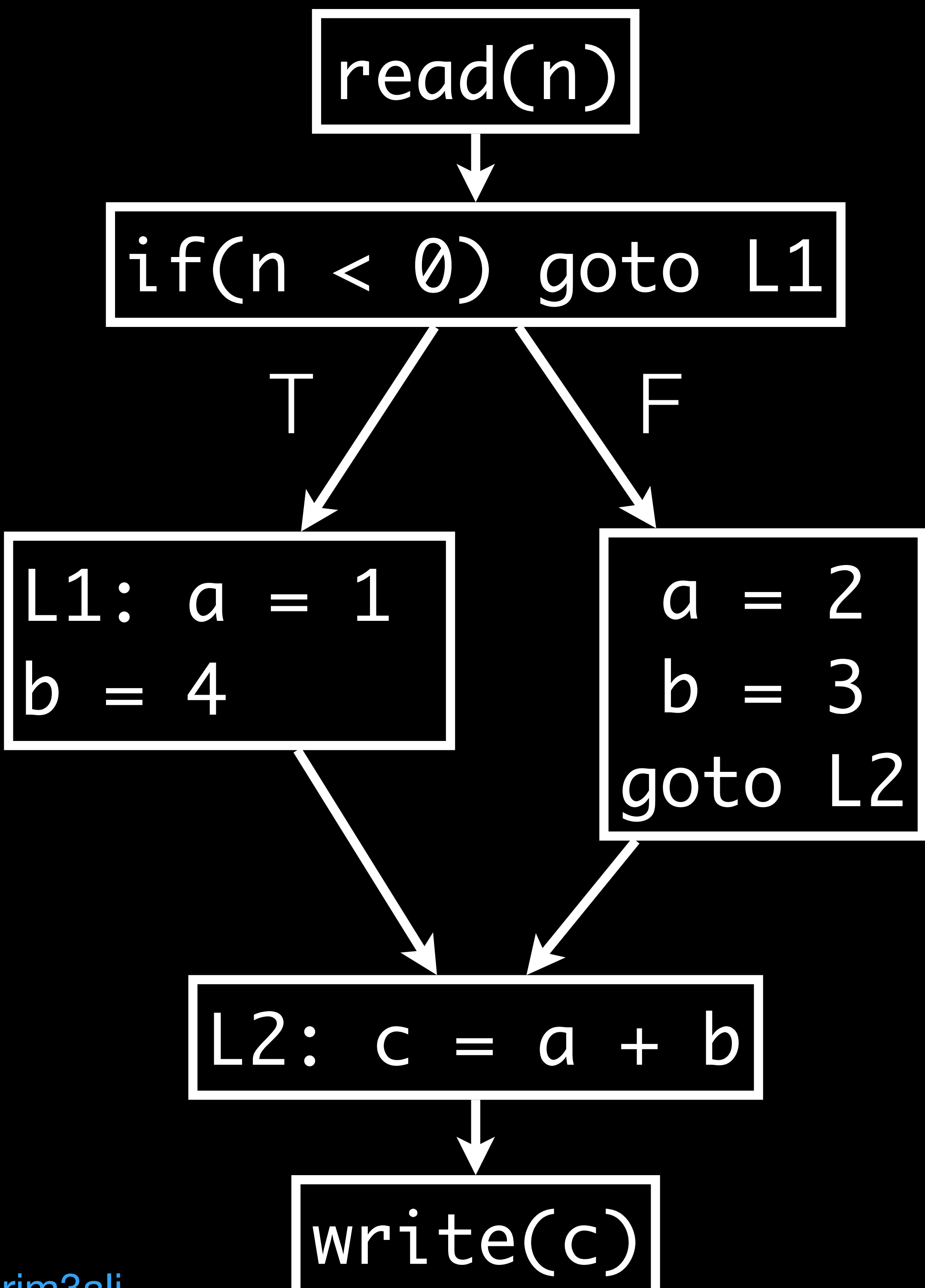
$$f_{write(c)}(f_{c\,=\,a+b}(f_{b\,=\,4}(f_{a\,=\,1}(f_{n\,<\,0}(f_{read(n)}(init))))))$$

Another path



$$f_{write(c)}(f_{c\,=\,a+b}(f_{b\,=\,3}(f_{a\,=\,2}(f_{n\,<\,0}(f_{read(n)}(init))))))$$

Paths Summary



$$f_{write(c)}(f_{c\,=\,a+b}(f_{b\,=\,4}(f_{a\,=\,1}(f_{n\,<\,0}(f_{read(n)}(init))))))$$

$$\sqcup$$

$$f_{write(c)}(f_{c\,=\,a+b}(f_{b\,=\,3}(f_{a\,=\,2}(f_{n\,<\,0}(f_{read(n)}(init))))))$$

Some Definitions

Partially-Ordered Set (poset)

- A set with the binary relation $\sqsubseteq$ that is:
  - reflexive ($x \sqsubseteq x$),
  - transitive ($x \sqsubseteq y \wedge y \sqsubseteq z$) $\implies x \sqsubseteq z$, and
  - anti-symmetric ($x \sqsubseteq y \wedge y \sqsubseteq x$) $\implies y == x$

Poset Upper Bound

- z is an **upper bound** of x and y if $x \sqsubseteq z$ and $y \sqsubseteq z$
- z is a **least upper bound** of x and y if:
  - z is an upper bound of x and y, and
  - for each upper bound v of x and y, $z \sqsubseteq v$.

Lattice

- A lattice is a poset such that for every pair of elements *x*, *y*, there exists:
  - a least upper bound ($x \sqcup y$) join
  - a greatest lower bound ($x \sqcap y$) meet

Lattice

- A lattice is **complete** if $\sqcup$ and $\sqcap$ exist for all (possibly infinite) subsets of elements
- A lattice is **bounded** if it contains 2 elements:
  - $\top$ (top) such that $\forall_x \, x \sqsubseteq \top$,
  - $\bot$ (bottom) such that $\forall_x \, \bot \sqsubseteq x$.
- **Q**: a complete lattice is bounded?
- **Q**: a finite lattice is complete?

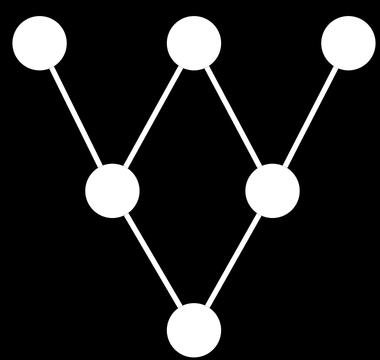
Lattice

- A **chain** is a set C of elements such that for all $x, y \in C$, $x \sqsubseteq y$ or $y \sqsubseteq x$.
- The **height** of a lattice is the cardinality of the longest chain
- In static analysis, we are interested in figuring out whether that height is finite or not!

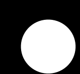
Types of Lattices

- **Powerset Lattice**: if F is a lattice, then the powerset $\mathcal{P}(F)$ with $\sqsubseteq$ defined as $\subseteq$ (or as $\supseteq$) is a lattice.
- **Product Lattice**: if L_A and L_B are lattices, then their product $L_A \times L_B$ with $\sqsubseteq$ defined as $(a_1, b_1) \sqsubseteq (a_2, b_2)$ if $a_1 \sqsubseteq a_2$ and $b_1 \sqsubseteq b_2$ is also a lattice.
- **Map Lattice**: if F is a set and L is a lattice, then the set of maps $F \rightarrow L$ with $\sqsubseteq$ defined as $m_1 \sqsubseteq m_2$ if $\forall_{f \in F} m_1(f) \sqsubseteq m_2(f)$ is also a lattice.

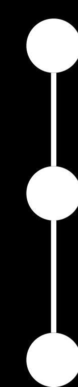
Are these lattices?



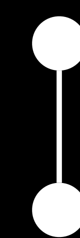
(A)



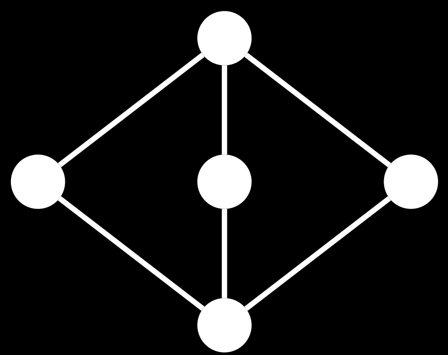
(B)



(C)



(D)



(E)

putting it all together!

Dataflow Framework

- For each statement S in the control-flow graph, define a $f_S : L \rightarrow L$.
- For a path $P = S_0 S_1 S_2 \ldots S_n$ through the control-flow graph, define $f_P(x) = f_n(\ldots f_2(f_1(f_0(x))))$.
- Goal: find the meet-over-all-paths (MOP)

$$MOP(n, x) = \bigsqcup_{P \text{ is a path from } S_0 \text{ to } S_n} f_P(x)$$

undecidable

[Kam, Ullman 1977]

Dataflow Framework

- For each statement S in the control-flow graph, define a $f_S : L \to L$.
- Goal: for each statement S in the control-flow graph, find $V_{Sin} \in L$ and $V_{Sout} \in L$ satisfying

Least-Fixed-Point (LFP)

MOP(n, x) $\sqsubseteq$ LFP(n, x)

$$V_{Sout} = f_S(V_{Sin})$$

$$V_{Sin} = \bigsqcup_{P \in \text{Predecessors}(S)} V_{Pout}$$

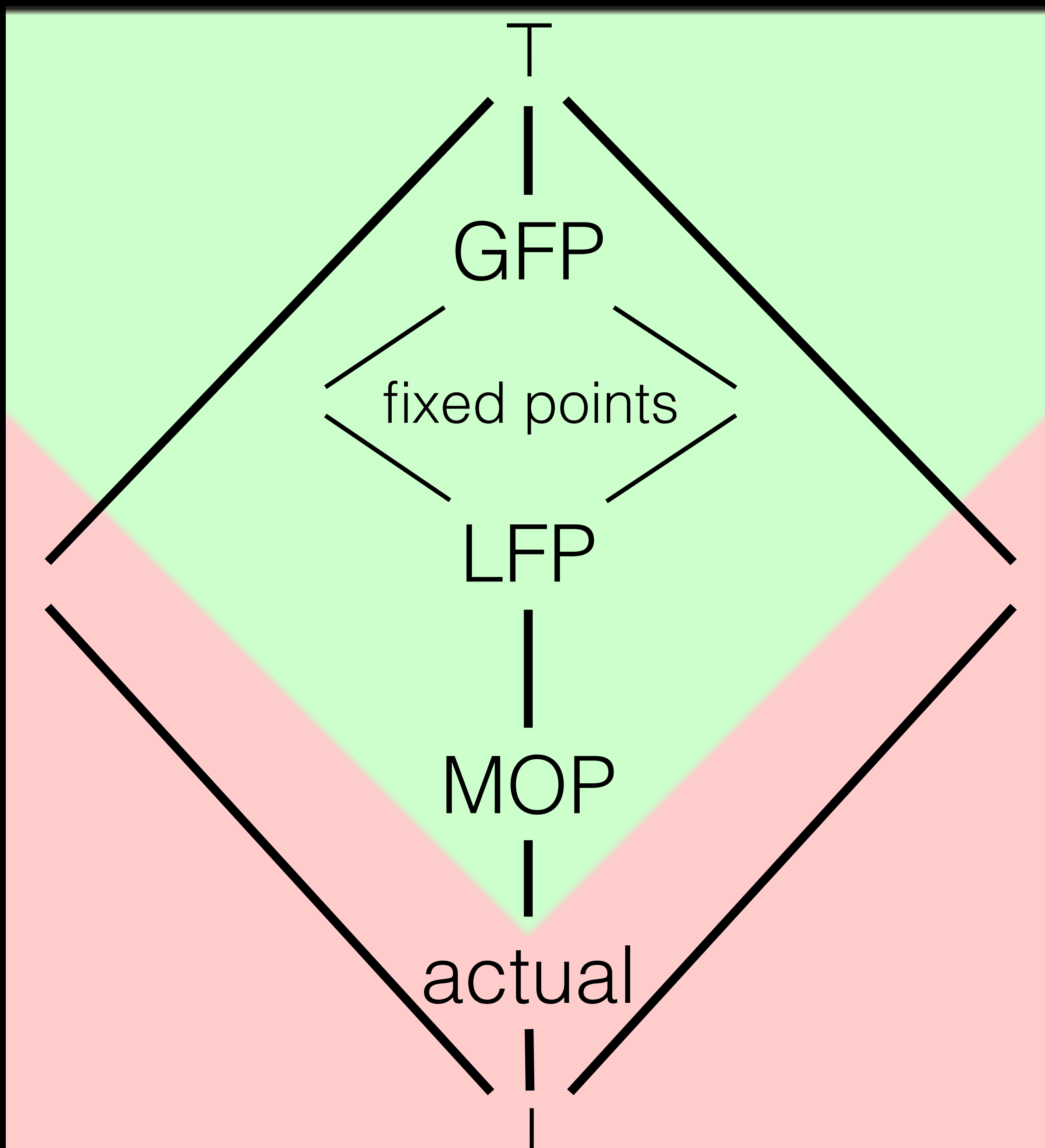
Generic Dataflow Algorithm

```
initialize out[s] = in[s] = ⊥ for all s
add all statements to worklist
while worklist not empty
  remove s from worklist
  in[s] = ⊔ p∈PRED(s) . out[p]
  out[s] = f_s(in[s])
  if out[s] has changed
    add successors of s to worklist
  end if
end while
```

Designing a Dataflow Analysis

1. Forwards or backwards?
2. What is the domain of the analysis info (lattice elements)?
3. What's the effect a statement has on the info? (flow functions)
4. What values hold at program entry points?
5. What's the initial estimate? It's the unique element $\perp$ such that $\forall_x \perp \sqcup x = x$.
6. How to merge info? (join/merge operator)

MOP ⊑ LFP



- Every solution S ⊒ *actual* is "safe" (i.e., sound).
- MOP ⊒ *actual*
- LFP ⊒ MOP
- A flow function *f* is distributive if *f*(x) ⊔ *f*(y) = *f*(x ⊔ y)
- If all flow functions are distributive, then LFP = MOP
- Initializing using ⊤ instead of ⊥ causes earlier termination, but yields more imprecise fixed-point

On Thursday

- Call graphs